MIT/LCS/TR-361

BOUNDED WIDTH BRANCHING PROGRAMS

David A. Barrington

May 1986

BOUNDED WIDTH BRANCHING PROGRAMS

by

DAVID ARNO BARRINGTON

B.A., Amherst College (1981)

C.A.S., Cambridge University (1982)

SUBMITTED TO THE DEPARTMENT OF MATHEMATICS
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1986

Signature of Author David A. Barrington ....

Department of Mathematics
May, 1986

Certified by Michael Sipser ....

Professor Michael Sipser
Thesis Supervisor

Accepted by ....

Professor Nesmith C. Ankeny, Chairman
Departmental Committee on Graduate Students
Department of Mathematics

BOUNDED WIDTH BRANCHING PROGRAMS

by

DAVID ARNO BARRINGTON

Submitted to the Department of Mathematics
on May 2, 1986 in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract:

We examine the branching program model of computation and in particular the classes of languages which can be recognized when the width of the programs is bounded by a constant. After slightly revising the framework of definitions to sharpen analogies with other models, we prove that width 5 polynomial size branching programs can recognize exactly the parallel complexity class NC^1, refuting a conjecture of Borodin et al. in [BDFP83]. Other results include an application to Boolean circuits of constant width (again, width 5 and polynomial size circuits can recognize exactly NC^1) and a characterization of a restricted class of width 3 branching programs. This thesis contains the results of [Ba85] and [Ba86], along with some additional material.

Thesis Supervisor: Dr. Michael Sipser

Title: Associate Professor of Applied Mathematics

Acknowledgements

My Ph.D. research was done under the direction of Mike Sipser, who suggested this topic and advised me throughout my work on it. I would like to thank him for all his help and also thank Ravi Boppana, Johan Hastad, David Johnson, Philip Klein, Tom Leighton, and Leslie Valiant for various helpful discussions.

This thesis was prepared using the M.I.T. Mathematics Department's Xerox Sun network and Leslie Lamport's LaTeX macros for Donald Knuth's TeX typesetting system. I would like to thank system manager Jon Haass, system guru Bob Thurstans, and local LaTeX guru Steve Lowe for their invaluable assistance.

My paper [Ba86] is a copyrighted work of the Association for Computing Machinery, and I use material from it herein by their permission.

This work was financially supported by National Science Foundation grant MCS-8304769, by U.S. Air Force grant AFOSR-82-0326, and by a National Science Foundation Graduate Fellowship. Vital emotional and spiritual support was provided for the author by his fiancé, Jessica Mix, to whom this work is dedicated.

CONTENTS

1. Introduction 5

2. Previous Work 7

3. Definitions and Justifications 10

4. The Cook Framework and BWBP 16

5. PBP's and the Width 3 Analysis 21

6. Solvable PBP's and the Fine Structure of NC^1 27

7. The Width 5 Result and its Consequences 31

8. Boolean Circuits of Constant Width 33

9. Non-solvable PBP's and Completeness 36

10. Uniformity 38

11. Open Problems 41

12. References 43

1. Introduction

Branching programs are a model of computation intermediate between parallel and sequential computation. The settings of n input variables determine a flow of control through an array of processors, as each processor triggers one of two successors depending on the value of one of the inputs. Originally invented for the analysis of switching problems, they have come to be analyzed as an abstract model.

The bounded-width problem was introduced by Borodin et al. in [BDFP83]. They proposed to develop a lower bound technology for general branching programs by first considering the case where the width of the processor array was bounded by a constant. This defined the complexity class which I call $BWBP$, languages recognizable by families of branching programs of constant width and polynomial size. They conjectured that the majority problem was not in $BWBP$, and provided a framework for analyzing branching programs of width two. Results in their paper and several suceeding papers seemed to support their conjecture — these are summarized in more detail in Section 2 below.

I began my analysis of bounded-width branching programs with their general program in mind, but I soon developed a different conceptual framework in which I changed the definitions of width and other facets of the model in ways that preserved the fundamental class $BWBP$. I think of a branching program as a series of instructions, each associating to one input variable two functions on a finite set with size equal to the width. This new framework makes more exact an analogy between bounded-width branching programs and a certain class of non-uniform finite automata. It also allows the introduction of notions from the theory of permutation groups which have made possible the results of this thesis. This conceptual framework and the ensuing definitions are described in Section 3.

In Section 4 I consider the position of the class $BWBP$ relative to other more well-known complexity classes. The context in which I do this is the hierarchy of parallel complexity clases described by Cook [Co85], which I describe and motivate. I prove that $BWBP$ lies within the class NC^1 and properly contains the class AC^0.

Given the redefinition of bounded-width branching programs as sequences of instructions consisting of finite functions, a natural notion is to restrict these functions to being permutations of the finite set. In fact for any group G realized as a permutation group, we can define a set of G-permutation branching programs (G-PBP's) where all functions are restricted to be permutations in G. If G is the symmetric group S_w we speak of width w permutation branching programs or w-PBP's. In Section 5 I characterize languages recognizable by 3-PBP's, and in Section 6 I extend part of this analysis to give strong evidence that G-PBP's for any solvable G have only limited computational power. This will involve a study of the structure of the complexity class NC^1 under AC^0 reductions.

In Section 7 I prove the main result of this thesis, that polynomial size 5-PBP's can recognize exactly those languages in NC^1. The consequences of this result for branching programs and NUDFA's are examined. In Section 8 a model of constant-width Boolean circuits is defined which is closely related to bounded-width branching programs — the relationship is proved and the consequences of the main result are examined. An extension of the main argument in Section 9 shows that polynomial-size G-PBP's can recognize all of NC^1 for any non-solvable G, so that solvability appears to be the key property.

The results discussed so far have concerned classes of non-uniform families of branching programs and Boolean circuits. In Section 10, using the definitions of Cook [Co85] and Ruzzo [Ru81], I show that the main result that $BWBP = NC^1$ also holds in a uniform setting.

Finally, in Section 11 I examine the possible applications of these results and the many open problems suggested by this research.

2. Previous Work

One general program in the study of computational complexity has been to take general models of computation, such as Turing machines or Boolean circuits, and examine their behavior under very tight resource constraints. This leads to the definition of relatively small 'low-level' complexity classes which lie inside those of primary interest. A prime example is the class AC^0 of problems solvable by Boolean circuits of constant depth, polynomial size, and unbounded fan-in. This class has been the subject of extensive research since the seminal paper of Furst, Saxe, and Sipser in 1981 [FSS81].

The study of these classes has a number of purposes. In some cases the possibility or impossibility of performing tasks under tight constraints is of practical interest. The work of [FSS81] rigorously confirmed a long-standing folk belief as to the impossibility of multiplying integers with a programmable logic array.

There have been direct mathematical connections to the study of the polynomial time complexity classes — for example, [FSS81] and Sipser in [Si83] showed that certain lower bounds for circuits of constant depth and unbounded fan-in implied the existance of oracles under which widely held conjectures about the polynomial time hierarchy are true. Now that these lower bounds have been provided by Yao [Ya85] and Håstad [Hå86], we know that the hierarchy can be separated from polynomial space by an oracle and that another oracle separates the individual levels of the hierarchy.

Finally, these classes appear to be more amenable to combinatorial analysis, so that we can discover things about them which will advance our knowledge of complexity classes in general, and develop new techniques which may have wide applicability. For example, the study of size complexity for Boolean circuits will be greatly affected by the recent work of Razborov [Ra85, Ra85a] and Andreev [An85] on monotone Boolean circuits. We will not discuss this work here — the reader should consult Johnson's excellent survey article [Jo86] for a description of both this and the oracle results, and further pointers to the relevant literature.

Our subject here is the branching program model of computation and in particular the effect upon it of tight constraints on width. As we shall see, complexity classes defined by branching programs fit closely into the framework of already studied low-level classes defined by Turing machines or circuits. It is hoped that the results here will further the general program of low-level complexity theory.

Branching programs were defined by Lee [Le59] as an alternative to Boolean circuits in the description of switching problems — he called them 'binary decision programs'. They were later studied in the Master's thesis of Masek [Ma76] under the name of 'decision graphs'.

Borodin, Dolev, Fich and Paul [BDFP83] raised the question of the power of bounded-width branching programs. They noted that the class $BWBP$ contains AC^0 (languages recognized by unbounded fan-in, constant-depth, polynomial-size Boolean circuits) as well as the parity function (shown to be outside AC^0 in [FSS81] and [Aj83]). They conjectured that the majority function was not in $BWBP$, in fact that for bounded width it requires exponential length.

Subsequent results appeared to support this conjecture. Chandra, Furst, and Lipton [CFL83] and Púdlak [Pú84] showed linear and superlinear length lower bounds respectively for arbitrary constant width. In [BDFP83] the idea was to work with width 2 and get exponential bounds. They succeeded for a restricted class of BP's, and Yao [Ya83] followed with a superpolynomial lower bound for general width 2. Shearer [Sh85] proved an exponential lower bound for the mod 3 function with general width 2. Ajtai et al. have just proved a nearly $n \log n$ size lower bound for a large class of symmetric functions [ABHKST86], where width is unconstrained but size is defined to be length times width.

Barrington [Ba85] revised the notion of width to the one used here and considered width 3 permutation branching programs. Their power was characterized as equal to that of certain depth 2 circuits of mod-2 and mod-3 gates, and it was shown that these could recognize any set in exponential length and that exponential length was required to recognize a singleton set.

Finally, in [Ba86] it was shown that the majority function (along with the rest of the class NC^1) is in $BWBP$, and thus that the conjecture of [BDFP83] is false.

This thesis contains the results of [Ba85] and [Ba86], along with some additional material.

3. Definitions and Justifications

The original notion of a branching program is that of a directed graph with decision nodes, accepting nodes, and rejecting nodes. To define the width of a branching program, we follow the process in [BDFP83]. We first divide the nodes of the graph into levels, i. e., sets $L_1, \ldots, L_k$ such that all edges out of nodes in L_i go to nodes in some L_{i+1}. We can make a graph levelled by adding more nodes, possibly squaring the size but keeping the length the same. The width is then the size of the largest level in an optimal division into levels.

An arbitrary graph of width w and length l can be converted into a w by l array of nodes by adding dummy nodes, possibly multiplying the size by w. This brings us to the model of [BDFP83].

Clearly a sequential computation with k internal states and running time t (where one time step is required to access an input variable) may be simulated by a branching program of width k and length t. This gives simulations of deterministic finite automata (DFA's) in constant width and log-space Turing machines in polynomial width and hence polynomial size.

In fact polynomial-size branching programs are equivalent to log-space Turing machines except for the problem of uniformity. A language $A \subseteq \Sigma^*$ can be recognized by a family of polynomial-size branching programs iff it can be recognized by a log-space Turing machine with polynomial *advice*. That is, along with the input $\mathbf{x}$ of size n the Turing machine receives a string $\mathbf{y}_n$, of size polynomial in n. The class thus defined is called 'non-uniform L', just as the languages definable by polynomial-size Boolean circuits are 'non-uniform P'. Given our later definition of width for Boolean circuits in Section 8, non-uniform L may also be defined as those languages computable by Boolean circuits of width $O(\log n)$ and polynomial size, as mentioned in [Jo86].

There are two approaches to dealing with this analogy — we may work in a uniform or a non-uniform setting. In the uniform approach we would define uniform families of branching programs, where the branching program for inputs of length n can be determined from n with appropriate limits on computational power. We will do this later when we discuss the parallel complexity classes such as NC^1. For now, we will consider arbitrary families of branching programs satisfying constraints of width, length, or size.

Our changes in the definitions of [BDFP83] are motivated by a desire to strengthen the relationship between branching programs and sequential computation, using an analogy of time for length and machine state for width. For example, consider an accepting node in the middle of a branching program. The flow of control through the program stops and never reaches the end. But if we are to think of the length dimension of the program as time, we must deal with the fact that at the end the fact of acceptance is known without having been stored in the meantime. A machine simulating the program would have to go into a separate 'accepting node encountered' state, and it seems reasonable to charge for this state in extra width.

Thus we require that accepting and rejecting nodes occur at the end, and note that we can impose this restriction on an arbitrary [BDFP83] branching program at a cost of adding two to the width without changing the length.

Next, we consider how the branching program is to get its input. As things stand, at a given level it can access one of several different input bits at the same 'time', depending on the 'state'. We can eliminate this power by insisting that all the nodes on the same level access the same input variable. A straightforward argument shows that we can impose this restriction at a cost of doubling the width and multiplying the length by either the width or by n, the number of input variables.

Now we can think of levels, rather that nodes, as our fundamental objects, and give the following formal framework for branching programs of bounded width.

Definitions: A *branching program* of *width* w (a w-BP) is a series of *instructions* $\langle j_i, f_i, g_i \rangle$ for $1 \leq i \leq l$, where x_{j_i} is one of n input variables, f_i and g_i are functions from $[w]$ to $[w]$ (here and throughout, $[w]$ is the set $\{0, \ldots, w-1\}$), and l is the length. Given a setting $\mathbf{x}$ of the input variables, the instruction $\langle j_i, f_i, g_i \rangle$ *yields* the function f_i if x_{j_i} is on and g_i if x_{j_i} is off. A branching program B yields the composition of the functions yielded by its instructions — we call this composition $B(\mathbf{x})$. Thus $B(\mathbf{x})$ applied to i gives the number of the sink reached by following the flow of control from node i in the first level given input setting $\mathbf{x}$.

There is now a choice in how to define the recognition of a language by a branching program. To match the earlier definition we should give a partition of $[w]$ into an accepting and rejecting set and say that B accepts $\mathbf{x}$ iff $B(\mathbf{x})$ applied to 0 is in the accepting set. For various reasons, we will use a number of definitions of acceptance below, explaining each when necessary.

It should be noted that all of our modifications of the definitions have preserved the class $BWBP$ of languages recognized by families of branching programs of constant width and size polynomial in n. We will eventually show that this class is independent of the definition of recognition — for now we may adopt the one given above.

We can now also formalize our notion of a non-uniform DFA and prove the relationship between such machines and bounded-width branching programs.

Definition: A *non-uniform deterministic finite automaton* (an NUDFA) is a machine with a two-way read-only input tape, a finite control with k states, and a one-way read-only *program tape.* On the program tape are instructions of two types — to move the input read head one position or to change machine state based on the current state and the input bit being read.

The automaton's resources are time (the length of the program tape) and space (its number of states, a constant). Note that the automaton must execute the entire program tape and that the input cell read at a given time does not depend on the input. These restrictions prevent the NUDFA from storing information about the input in the input read head position.

A width k braching program can easily simulate an k-state NUDFA program, with one branching program instruction for each state-changing NUDFA instruction. The simulation in the other direction is also easy, except that the NUDFA time may be as much as n times greater than the branching program length because the NUDFA may have to move its input head up to n times between branching program instructions. We have the same varied possibilities for how to define acceptance by an NUDFA — the most natural first notion is to designate a start state and partition the final states into accepting and rejecting.

This analogy allows us to prove facts about branching programs using standard arguments about finite automata. For example, we can define a non-deterministic branching program of width w by allowing more than one function for the on-case and off-case in each instruction. Using the familiar subset construction for DFA's and NFA's it is easily shown that a non-deterministic branching program of width w and length l may be simulated by a deterministic program of width 2^k and length l. Thus the class $BWBP$ could equivalently be defined using non-deterministic programs. When we show later that $BWBP$ is a subset of the parallel complexity class NC^1, we will use a simple adaptation of the standard argument that regular languages are in NC^1. The utility of the analogy, in my view, provides a compelling justification for the changes I have made in the definition of width.

The view of branching programs as analogous to DFA's makes clear the intuition behind the conjecture of [BDFP83] that the majority function is not in $BWBP$. The finite control and memory of an NUDFA are very simple, and if the number of states is small enough we might imagine an implementation of an NUDFA using a chimpanzee [1] as central processor, as follows. The memory (machine state) will

[1] We take no position here in the raging debate on chimpanzee intelligence. The defining properties of the chimpanzee in this discussion will be a capacity for symbolic manipulation with a constant sized vocabulary, together with an inability to think abstractly as humans do about such things as large integers. The zoologically informed reader has the option of positing a hypothetical animal with these properties if a chimpanzee will not do.

be stored as a single abstract symbol from a vocabulary of k such symbols known to the chimpanzee. The input will be a sequence of n cells on a tape, colored black or white, and accessible to the chimpanzee only one cell at a time through a window. The chimpanzee is able to move the window one cell at a time in either direction. The program tape of the NUDFA is now a series of instructions in a symbolic language known to the chimpanzee. The two types of instructions are represented by the examples 'move the input head left' and 'if the state is a and input is black change the state to b' (where a and b are symbols in the chimpanzee's vocabulary). It should be clear that if the chimpanzee executes the instructions infallibly, this model is mathematically equivalent to the NUDFA model we descibed earlier. Thus a language is in $BWBP$ iff there is a polynomial length family of instruction sequences to a chimpanzee with constant size vocabulary which causes the chimpanzee to recognize the language.

If majority is in $BWBP$, then there is a polynomial length sequence of instructions which will enable the chimpanzee to count the black squares in the input. But, the argument would go, the chimpanzee has only a constant amount of external memory and is assumed to be incapable of counting on his own! It is true that he can count in exponential length (as we shall see below), but this is done by checking each of the 2^n possible cases for the input and giving the output corresponding to the correct case. Majority, using this method, is no easier than any other Boolean function. The intuition behind the conjecture is that there can be no method of counting which is substantially better than this one and can be carried out by the chimpanzee.

The NUDFA model raises some interesting foundational questions. In a Turing machine, all the problem-solving capacity is stored in the finite control and its ability to manipulate the memory — the program is of constant size. Now we ask about a long program and a very limited machine, and compare this to other models.

4. The Cook Framework and BWBP

One of the principal questions in theoretical computer science in recent years has been the difference in power between sequential computation, such as that of a Turing machine, and parallel computation, as is contemplated in new types of machines. What problems can be solved much more quickly by the cooperative simultaneous action of a large number of processors? This has led to the development of new complexity classes of problems solvable by parallel algorithms under various resource constraints. Though bounded-width branching programs appear to be an essentially sequential model, the class *BWBP* turns out to have interesting relationships with these new parallel classes.

We will consider the hierarchy of complexity classes described by Cook in his survey article [Co85], which covers his own and others' research over the last decade. As he explains, there are a number of competing abstract models for parallel computation, but to develop an abstract complexity theory there is a strong case for the model of Boolean circuits. They are simple, offering a greater promise for the use of combinatorial techniques in proving lower bounds, and yet they effectively simulate the other models so that complexity results carry over into them.

A Boolean circuit is a directed acyclic graph whose nodes, or *gates*, calculate the and-function or or-function of their input edges and pass the result along their output edges. Input to the circuit is a fixed number n of bits which may be accessed individually from special nodes which output the value of an input variable or its complement. Sinks in the graph are output nodes, of which there may be one (in which case the circuit recognizes a subset of $[2]^n$, i.e., $\{0,1\}^n$) or several (in which case it calculates a function). The important parameters of a circuit are *size* (number of nodes), *depth* (length of the longest path from an input to an output, and *fan-in* (the maximum in-degree of a gate). A circuit operates only on inputs of a given size, so we define *families* of circuits and treat size and depth as functions of n. We will assume the reader to have a good understanding of this model — appropriate references may be found in [Co85].

The fundamental parallel complexity class, NC, is defined as those functions computable by Boolean circuit families with polynomial size and depth bounded by a polynomial in $\log n$. When we examine the degree of this polynomial to make finer distinctions among functions in NC, we must begin to consider the fan-in of the circuits.

Definition: For each natural number i, NC^i is defined to be those functions computable by polysize families with depth $O(\log^i n)$ and fan-in two. AC^i is defined similarly, but with no restriction on fan-in. (Note the remarks on uniformity below.)

Clearly a gate with arbitrary fan-in may be simulated by a binary tree of gates with fan-in two, and since the fan-in in an AC^i circuit is bounded by the size this tree has depth $O(\log n)$ and thus $AC^i \subseteq NC^{i+1}$.

To properly define these complexity classes we should define a uniformity condition, insisting that the circuit in a family which takes n inputs be computable from n with appropriate resource constraints. For now, however, we will allow our families of circuits, as well as our families of branching programs, to be non-uniform. Our main result remains true in the uniform setting of [Co85], as we will demonstrate later in Section 10.

Given an appropriate uniformity definition, the classes NC^i and AC^i have a wide applicability in the study of parallel computation. For example, each has an equivalent definition in terms of alternating Turing machines — NC^1 is those functions computable on an ATM with space $O(\log n)$ and time $O(\log^i n)$ [Ru81], while AC^i is those computable with space $O(\log n)$ and alternation depth $O(\log^i n)$ [Co85]. Also, AC^i is the class of functions computable in time $O(\log^i n)$ and polynomially many processors on a SIMDAG, a type of parallel random-access memory computer where read and write conflicts are allowed and the lowest numbered processor succeeds in the case of a write conflict [CSV82].

Both deterministic and nondeterministic log space are easily seen to lie between NC^1 and AC^1, while all of NC is a subset of polynomial time. As in sequential complexity theory, proofs that any classes in the hierarchy differ are rare, but it is generally conjectured that all or nearly all are actually different. Cook[Co85] begins his survey of these classes with NC^1, but to examine a very constrained class like $BWBP$ we must begin at the very beginning.

NC^0 is simply those functions whose output bits each depend on only a constant number of input bits (for example, bitwise operations on Boolean vectors). AC^0 includes such things as binary integer addition or Boolean matrix multiplication, where the dependence of an output bit on all the input bits is very simple — an AC^0 circuit has only constant depth though it has unbounded fan-in. However, the parity function (which returns one iff the number of ones in the input is odd) cannot be computed in AC^0, as was proved by Furst, Saxe, and Sipser [FSS81] and independently Ajtai [Aj83]. Many other functions can be shown to be outside AC^0 using this result and the technique of AC^0 reduction — one shows that an AC^0 circuit for the desired function could be used to construct one for parity.

We can begin to place $BWBP$ in relation to these classes by showing that it contains the languages in AC^0. An unbounded fan-in circuit of depth d and size s can be simulated by a branching program (using our definitions rather than those of [BDFP83]) of width $d+1$ and size s^d. We sketch the proof, which is by induction on d. The circuit of depth d is the AND or the OR of at most s circuits of depth $d-1$. We concatenate the branching programs for each of these subcircuits and add a new row of nodes (increasing the width by one) which the program will branch to iff the top node of the circuit is satisfied. We arrange that it will remain on this row if it ever gets there, so that at the end it is on that row iff the circuit is satisfied.

We come next to the class NC^1 of functions computable by circuits with fan-in two and depth $O(\log n)$ (the restriction to polynomial size is made redundant by these two). Parity is easily computable in NC^1 (by a binary tree of circuits for binary exclusive or), so we know that this class strictly contains AC^0. In fact quite a lot else is in NC^1 — integer multiplication, integer matrix multiplication, or any function which is symmetric (i.e., where the output depends only on the number of ones in the input). These functions may all easily be calculated using iterated integer addition as a subroutine (e.g., adding n numbers of n bits each), and the latter may be done in NC^1 by a binary tree of NC^0 circuits which add integers using a redundant notation — see [BCP83] for details. If we allow the circuits to be less uniform then integer division and several related functions are also in NC^1 — polynomial-time uniform circuits for these are given in [BCH84]. Non-uniform NC^1 may also be characterized as containing exactly those languages recognizable by families of polynomial length Boolean formulas (or, equivalently, of polynomial size circuits which are trees)[Sp71].

NC^1 has long been known to contain all regular languages, and we can exploit the analogy between finite automata and $BWBP$ to obtain the following proof that $BWBP$ is contained in NC^1 as well. This is simply a non-uniform version of an argument which in the uniform case essentially appears in [Sa72] and is given explicitly in [LF77].

Theorem 1: If $A \subseteq [2]^n$ is recognized by a w-BP B of length l, A is recognized by a fan-in 2 circuit of depth $O(\log l)$, where the constant depends on w.

Proof: We may choose the weakest possible notion of recognition here and say that B accepts $\mathbf{x}$ if $B(\mathbf{x})$ is in some arbitrary subset of the functions from $[w]$ to $[w]$. We can represent such a function f by w^2 Boolean variables telling whether $f(i) = j$ for each i and j. The composition of two such functions so represented may be computed by a fixed circuit whose size depends only on w. Our circuit for A will have a constant-depth section to find the function yielded by each instruction of B, a binary tree of composition circuits, and a constant-depth section at the top to determine acceptance given the function yielded by B.

We have thus seen that $BWBP$ is a subclass of NC^1 which includes the subclasses of AC^0 languages and regular languages. Given the 'chimpanzees can't count' intuition which suggests that $BWBP \neq NC^1$, we might think of $BWBP$ as the easier problems within NC^1 and thus an interesting proper subclass of it.

5. PBP's and the Width 3 Analysis

A logical program for studying $BWBP$ would be to begin with specific small constant bounds on width and investigate what can and cannot be done under them. This is the program proposed in [BDFP83] for their different definitions and we begin by retracing their steps. They found much worth studying in width 2, but under our definitions width 2 is very weak indeed (it is similar to their class SW_2). Most languages can be shown unrecognizable by 2-BP's with the aid of the following result.

Proposition: A language is recognizable by a width 2 branching program iff it is recognizable by such a program of length $O(n^2)$.

Proof: Of the sixteen possible instructions of a 2-BP, four set the yield to 0, 1, x_i, or $\bar{x}_i$ and destroy all previous information. The others change the function f calculated so far in ways which can be built up from the operations changing f to $\bar{f} = f \oplus 1$, $f \oplus x_i$, $f \wedge x_i$, or $f \wedge \bar{x}_i$ for an input x_i. We may delete any but the last occurrence of each of the $2n$ distinct AND instructions, because if their variable is false the last one destroys all previous information, and if it is true they all have no effect. Between two AND's, we need have only one XOR for each of the n variables and one NOT. (This idea of this proof is taken from [BDFP83].)

When we simulated AC^0 circuits by branching programs above, we used a general technique for simulating depth d unbounded fan-in circuits by width $d+1$ branching programs. This allows us to compute any Boolean function in width 3 and exponential length, using its conjunctive normal form circuit. No lower bounds are yet known for width 3.

Lower bounds can be derived, however, for a restricted class of 3-BP's, which we now define for width w as they will become important later. This class will turn out to be more susceptible to analysis because we will be able to bring in the theory of permutation groups.

Definition: A *permutation branching program* of width w (a w-PBP) is a w-BP where both the functions f_i and g_i in each instruction are permutations of $[w]$.

An unrestricted w-BP can be thought of as a series of w-PBP's linked by special instructions which coalesce two or more of the w 'machine states' into one. It is not immediately clear whether these special instructions are of any use at all, or are of vital importance to the w-BP. This is somewhat reminiscent of Bennett's work on Turing machines which do not destroy information during their computations [Be73]. In any case, we can begin to get a better picture of this by a more detailed analysis of the width 3 case.

For 2-BP's we had no particular difficulty in defining acceptance of a string by a branching program, but now we must be more precise. We have the two notions of partitioning either the nodes themselves or the functions on the nodes into accepting and rejecting sets. Also, in working with 3-PBP's we are going to want to build up larger programs from smaller, and this will be far easier if each program can be guaranteed to have only two possible yields. For these reasons we offer the following definitions and technical results.

Definitions: A w-BP B ***weakly recognizes*** a set $A \subseteq [2]^n$ if there is a subset S of the functions from $[w]$ to $[w]$ such that $B(\mathbf{x}) \in S$ iff $\mathbf{x} \in A$. B ***strongly recognizes*** A if there are two fixed functions f and g such that $B(\mathbf{x}) = f$ iff $\mathbf{x} \in A$ and $B(\mathbf{x}) = g$ iff $\mathbf{x} \notin A$.

Proposition: If A is weakly recognized by a 3-PBP there is another 3-PBP, at most a constant factor longer, which always yields an even permutation and also weakly recognizes A.

Proof: We sketch this only. Given a permutation $\sigma \in S_3$, many permutations derived from it are guaranteed to be even, such as $\sigma\tau\sigma^{-1}\tau^{-1}$ for any fixed τ. One must show that given any subset of S_3 there is such a mapping from S_3 to A_3 (the even permutations) which takes that subset exactly onto a subset of A_3. This is tedious but not difficult.

Proposition: If A is strongly recognized by a 3-PBP it is also strongly recognized by a 3-PBP of the same length which yields the identity if $\mathbf{x} \notin A$ and another fixed permutation if $\mathbf{x} \in A$.

Proof: The original 3-PBP gives σ or τ — we change this to $\sigma\tau^{-1}$ or the identity by composing both permutations in the last instruction with τ^{-1}.

Proposition: If A is strongly recognized by a 3-PBP and σ is an even permutation other than the identity, there is a 3-PBP at most twice as long which yields σ for $\mathbf{x} \in A$ and the identity for $\mathbf{x} \notin A$.

Proof: We can change ν and the identity to $\tau\nu\tau^{-1}$ and the identity for any τ by altering the first and last instructions. This suffices if ν and σ are conjugates, i.e., unless ν is odd. If ν is odd, we can make σ as a product of ν and one of its conjugates.

We see from these facts that the function and node definitions coincide for 3-PBP's, as even permutations of [3] can be characterized by where they take a single element. We will now think of a 3-PBP, then, as giving a function from $[2]^n$ to [3], which will facilitate our analysis. We will obtain our best results for strong recognition, though weak recognition is the more natural model.

If each instruction of a 3-PBP yielded an even permutation, we could find out the entire yield by adding up the individual contributions mod 3. We can't do this in general, because the individual contribution may be odd. However, we can do something just as good with a little more information.

Consider S_3 as generated by two elements x and r with $x^2 = r^3 = e$ and $xr = r^2x$. (Here e is the identity.) Given the input we have a yield $x^{b_i}r^{c_i}$ for each instruction, and we want the product. We will assume, using the above facts, that this product is even (a power of r), so we want to find the number of r's contributed by each instruction. Note that we could if we wanted write $x^{b_i}r^{c_i}$ as $r^{-c_i}x^{b_i}$. If we choose the notation for each instruction correctly, we can have all the x's occurring in pairs and thus vanishing. In other words, the i'th instruction will contribute c_i r's if the number of x's contributed before it (i.e., $\Sigma_{j<i}b_j$) is even, and $-c_i$ if it is odd. Thus a combination of mod 3 and mod 2 questions can evaluate the 3-PBP — we now make this precise.

Definition: A *3-2 circuit* consists of a single mod 3 gate whose inputs are constant 1 gates or parity (mod 2) gates connected to inputs. Its output is the sum mod 3 of the number of 1 gates and the number of parity gates which are satisfied. Its size is the fan-in of the mod 3 gate.

Proposition: A 3-PBP of length l and guaranteed even yield may be simulated by a 3-2 circuit of size $O(l)$.

Proof: We use the reasoning above, except that we must show how to take the input into account. It suffices to give a constant number of constant and parity gates whose output mod 3 is a given parity function of inputs if a given input variable is on, and 0 otherwise. This is straightforward as $P \wedge x$ is the sum mod 3 of $2x$, $2P$, and $P \oplus x$.

Proposition: A 3-2 circuit of size s may be simulated by a 3-PBP of length $O(ns)$, where n is the number of input variables.

Proof: It is easy to make a 3-PBP of length at most n which yields either a 2-cycle or the identity depending a a given parity condition. By concatenating two such 3-PBP's with different outputs and the same parity condition, we get a 3-PBP which yields a given 3-cycle or the identity depending on the parity condition and thus simulates a single parity gate of the 3-2 circuit.

Theorem 2: The optimal 3-PBP length and the optimal 3-2 circuit size of any function from $[2]^n$ to $[3]$ differ at most by a multiplicative factor of $O(n)$.

Proof: Immediate from the above.

There are exactly 2^n different types of gate in a 3-2 circuit — the constant 1 gate and the gate taking the parity of $x \in A$ for any nonempty subset A of the n variables. These types may be indexed by the set $[2]^n$, and thus a circuit may be described by a function C from $[2]^n$ to $[3]$, with $C(A)$ being the number of gates of the type corresponding to the set $A \subseteq [n]$. (Without loss of generality, there are zero, one, or two such gates.) The output function D of the circuit is also from $[2]^n$ to $[3]$, where $D(B)$ for $B \subseteq [n]$ is the output when exactly those variables x_i with $i \in B$ are on.

If we view the functions from $[2]^n$ to $[3]$ as a vector space of rank 2^n over the field $GF(3)$, then the mapping from a circuit's description to its output function is linear, given by the matrix $M = \{m_{AB} : A \subseteq [n], B \subseteq [n]\}$ where $m_{AB} = 1$ if $A = \emptyset$ and $m_{AB} = |A \cap B| \bmod 2$ otherwise. This matrix has nonzero determinant over $GF(3)$ — to see this we look at a similar matrix M' where $m'_{AB} = 2 - (|A \cap B| \bmod 2)$. M' is derivable from M by Gaussian operations and thus has nonzero determinant iff M does. By explicit calculation one can easily show that the square of M' is $\pm I$.

Thus this mapping from circuit descriptions to functions is an isomorphism of the vector space $GF(3)^{2^n}$. Each function thus has a unique circuit calculating it, of size $O(2^n)$. We can thus find the circuit complexity of any specific function if we can find a general form for its inverse image under this mapping. In the case of the and-function ($D(B) = 1$ if $B = [n]$; $D(B) = 0$ otherwise) this inverse image is given by $C(A) = 2^n \cdot (2 - (|A| \bmod 3)) \bmod 3$, and so the circuit has size $\frac{3}{2} \cdot 2^n$.

Putting this together with the theorem above, we get:

Theorem 3: Every subset of $[2]^n$ may be strongly recognized by a 3-PBP of length $O(n2^n)$. Length $O(2^n)$ is required to strongly recognize the singleton set $\{1^n\}$, i.e., to strongly calculate the and-function of n variables.

Allowing weak recognition can help considerably. To weakly recognize $\{1^n\}$, we can concatenate two 3-PBP's which strongly calculate the and-functions of the first $n/2$ and last $n/2$ variables respectively. This 3-PBP outputs 2 iff all the variables are on, and has size $O(2^{n/2})$. We conjecture that this is optimal, but this seems very difficult to prove.

6. Solvable PBP's and the Fine Structure of NC^1

We have just shown that the languages strongly recognizable by 3-PBP's are not all of NC^1, because they do not include the and-function (so they are not even all of AC^0). We are going to generalize part of this argument to give strong evidence that many classes of PBP's cannot recognize all of NC^1 — to do this we must develop a theory of the internal structure of NC^1 as a complexity class. This will be a degree theory analagous to the degrees of unsolvability in classical recursion theory or the polynomial-time degrees within NP.

Inside NC^1, it is most natural to define AC^0 reductions — the function f is reducible to g (written $f \leq_{AC^0} g$) if a constant-depth poly-size unbounded fan-in circuit, containing oracle nodes for g, can compute f. If f and g are each AC^0 reducible to the other we say they are AC^0 equivalent, and the AC^0 degrees are the equivalence classes of this relation. The class AC^0 is itself a degree, and NC^1 is partitioned into it and one or more others.

AC^0 reducibility was introduced in [FSS81] under the name of 'cp-reducibility'. They suggested further study of the degree structure (they had only just given the first proof that the structure of NC^1 was non-trivial) and conjectured that majority was not reducible to parity.

Their suggestion was taken up by Fagin et al. in [FKPS84], who found many new AC^0 reducibilities among symmetric functions. Modulo the new parity lower bounds of Yao [Ya85] and Håstad [Hå86], they characterize those symmetric functions in AC^0. They show that the degree of the majority function is complete for symmetric functions and contains a large class of symmetric functions (though there seems to be no reason to believe that this degree is complete for NC^1). Interestingly, no complete symmetric function exists in the projection-reducibility theory of Valiant [SV81], by a recent result of Geréb-Graus and Szemerédi [GS??].

It is still not known, however, that there are more than two AC^0 degrees within NC^1. The conjecture that majority is not reducible to parity would settle this, as parity would then be in an intermediate degree between the majority degree and AC^0. This conjecture seems very plausible, as the ability to count mod 2 would not seem to help a circuit to count overall. We make the following conjecture, strengthening that of [FSS81]:

Conjecture: Majority is not AC^0 reducible to the mod k function for any k, and thus no mod k function is AC^0 complete for NC^1.

In our analysis of 3-PBP's, we saw that a constant depth circuit of mod 2 and mod 3 gates could determine the output of a 3-PBP, so that any language recognizable by a 3-PBP is AC^0 reducible to the mod 6 function. We shall now see that this part of the analysis depended on properties of the group S_3 from which we took permutations, in particular the *solvability* of that group. First we must generalize our definitions to arbitrary groups.

Definition: Let G be a finite group realized as a group of permutations of $[w]$. A *G-permutation branching program* (or G-PBP) is a w-PBP where both the permutations in each instruction are taken from G.

Definition: The *word problem* for a fixed group G is to input an ordered string of elements of G (using any fixed representation) and output their product.

Proposition: The problem of evaluating the output of a G-PBP given an input is AC^0 equivalent to the word problem for G.

Proof: Given a G-PBP and an input, simply read off the permutations to be composed. Given a sequence of elements of G, simply make a G-PBP where each instruction yields the corresponding element on any input, so the output is the product of the elements.

Now we will give one of the many equivalent definitions of solvability. (For more detail see a group theory text such as [Za58].) The commutator subgroup of G is the subgroup generated by all elements of the form $aba^{-1}b^{-1}$ for a and b in G. A group is solvable if and only if repeated taking of commutator subgroups eventually gives the trivial group. Thus a group is non-solvable if and only if it has a nontrivial subgroup whose commutator subgroup is itself. (All groups under discussion are finite.)

The following theorem extends the earlier argument to arbitrary solvable groups.

Theorem 4: The word problem for any fixed solvable group G is AC^0-reducible to the mod g function, where g is the order of G.

Proof: An equivalent definition of a solvable group (see, e.g., [Za58]) is one which has a series of normal subgroups $G = G_0, G_1, \ldots, G_m = \{e\}$ where each quotient group G_i/G_{i+1} is cyclic. We prove the theorem by induction on the length of this series. So assume that G has a normal subgroup N, where G/N is cyclic and the word problem for N is solvable by an AC^0 circuit containing mod g gates. Choose an element a such that the coset aN generates G/N.

We are given a product $g_1 \ldots g_k$ to evaluate. As N is normal, we can write each g_i uniquely as $a^{\epsilon_i} n_i$ with $n_i \in N$. (Converting between any two bit representations of an element of G takes constant size and depth.) Now let b_i be the product $a^{\epsilon_1} \ldots a^{\epsilon_i}$ and note that $a^{\epsilon_1} n_1 \ldots a^{\epsilon_k} n_k = (b_1 n_1 b_1^{-1}) \ldots (b_k n_k b_k^{-1}) b_k$. Each b_i depends only on the sum mod g of the appropriate ϵ_j, as the order of a in G divides g. Each term $b_i n_i b_i^{-1}$ is in N by normality, and we can calculate it in constant depth using mod g gates to get b_i. These partial terms may then be multiplied using a circuit for N.

Theorem 4 is interesting only if the Conjecture above is true. Proving that conjecture, however, will apparently need an entirely new method. Unfortunately, the random restriction method of [FSS81] does not seem to extend to even parity (mod 2) gates, as the restriction of a parity gate is still a parity gate.

Håstad [Hå86a] has recently proved a partial result toward the Conjecture. He shows that any constant depth and polynomial size circuit of AND, OR, and parity gates which computes majority must have $\Omega((\log n)^3/2)$ parity gates. His method appears to be inherently limited to circuits with fewer than n parity gates, and thus it appears that something new is still needed.

7. The Width 5 Result and its Consequences

We will now see that the view of branching programs as being composed of permutations allows us to prove our surprising main result. Unlike 3-PBP's and 4-PBP's (which fall under the above results because S_3 and S_4 are solvable groups), 5-PBP's can recognize all of NC^1 in polynomial size. We will state the result in an even stronger form to allow ourselves to carry out the necessary induction.

We say that a 5-PBP B ***five-cycle recognizes*** a set $A \subseteq [2]^n$ if there exists a five-cycle σ (called the ***output***) in the permutation group S_5 such that $B(\mathbf{x}) = \sigma$ if $\mathbf{x} \in A$ and $B(\mathbf{x}) = e$ if $\mathbf{x} \notin A$ (e is the identity permutation).

Theorem 5: Let A be recognized by a depth d fan-in 2 Boolean circuit. Then A is five-cycle recognized by a 5-PBP B of length at most 4^d.

Lemma 1: If B five-cycle recognizes A with output σ and τ is any five-cycle, then there exists a 5-PBP B', of the same length as B, which five-cycle recognizes A with output τ.

Proof: Since σ and τ are both five-cycles there exists some permutation θ with $\tau = \theta\sigma\theta^{-1}$. To get B', simply change each instruction of B, replacing each σ_i and τ_i by $\theta\sigma_i\theta^{-1}$ and $\theta\tau_i\theta^{-1}$.

Lemma 2: If A is five-cycle recognized in length l, so is its complement.

Proof: Let B five-cycle recognize A with output σ. Call the last instruction of B $\langle i, \mu, \nu \rangle$. Let B' be identical to B except for last instruction $\langle i, \mu\sigma^{-1}, \nu\sigma^{-1} \rangle$. Then $B'(\mathbf{x}) = e$ if $\mathbf{x} \in A$ and $B'(\mathbf{x}) = \sigma^{-1}$ if $\mathbf{x} \notin A$. Thus B' five-cycle recognizes the complement of A.

Lemma 3: There are two five-cycles σ_1 and σ_2 in S_5 whose commutator is a five-cycle. (The commutator of a and b is $aba^{-1}b^{-1}$.)

Proof: $(12345)(13542)(54321)(24531) = (13254)$.

Proof of Theorem 5: By induction on d. If $d = 0$ the circuit is an input gate, and A can easily be recognized by a one-instruction 5-PBP. Using Lemma 2 in the case of an OR gate, assume without loss of generality that $A = A_1 \cap A_2$, where A_1 and A_2 have circuits of depth $d-1$ and thus 5-PBP's B_1 and B_2 of length at most 4^{d-1}. Let B_1 and B_2 have outputs σ_1 and σ_2 as in Lemma 3, and B_1' and B_2' have outputs σ_1^{-1} and σ_1^{-1} (This last is possible by Lemma 1). Let B be the concatenation $B_1B_2B_1'B_2'$. B yields e unless the input is in both A_1 and A_2, but yields the commutator of the two outputs if the input is in A. This commutator is a five-cycle, and so B five-cycle recognizes A. B has length at most 4^d. Given a circuit and a desired output, this proof gives a deterministic method of constructing the 5-PBP.

This result has interesting consequences in the realm of NUDFA's – in particular, our earlier intuition appears to be wrong. A chimpanzee can be given a polynomial-length set of instructions which allow him to count, as well as compute any symmetric function of the input. In fact, if we allow ourselves a polynomial-time Turing machine to generate his instructions, he can also divide integers and compute the related functions of [BCH84].

8. Boolean Circuits of Constant Width

We define width for Boolean circuits so as to allow nodes at any level to access the inputs without penalty, and examine the consequences of our main result for constant-width circuits in this model. It is easy to show [Ho83] that constant width for branching programs is equivalent to constant width for circuits, but here we go into more detail in an attempt to get the best possible simulations.

In particular, we show that width w branching programs (using the definitions of [Ba86]) can be simulated by circuits of width $\lceil \log w \rceil + 1$ and length multiplied by a constant depending only on w (this is a slight improvement of a result of Hoover [Ho83], who simulated width w BP's in the [BDFP83] model by circuits of width $\lceil \log w \rceil + 4$.) In particular, width 5 branching programs can be simulated by width 4 circuits (improving the result cited in [Jo86]), so that width 4 polynomial circuits can recognize all of NC^1 and thus everything recognized by circuits of constant width and polynomial size.

We choose the following definition of a width-w circuit from the many equivalent ones. A circuit is a rectangular array of nodes, consisting of l rows of w nodes each. Each node has one or two edges entering it which must be from either inputs or nodes on the immediately previous row. Possible node types are EQUALS (unary), NOT (unary), AND (binary), and OR (binary). Edges carry Boolean values, and nodes send out the appropriate value calculated from their input or inputs.

This is equivalent to other definitions which allow wires (edges) to jump over intermediate levels but count them as part of the width for those levels. (See, for example, [Jo86].) Perhaps the most natural first definition of width would charge for access to the inputs, but this would lead to a class far too restricted to be interesting.

Note that for defining the class of functions calculable using width w and length $O(f(n))$, we have a lot of latitude in our definitions. We will think of the inputs as being accessed by unary AND-x_i, AND-$\bar{x}_i$, OR-x_i, or OR-$\bar{x}_i$ gates — any other use of x_i can be simulated by these in a constant number of rows. We will also assume that only one input variable is accessed by a given row of nodes — this can be enforced by replacing one row by up to w rows.

Proposition: A Boolean circuit of width w and length l may be simulated by a branching program of width 2^w and length w.

Proof: Use the 2^w nodes in each instruction to represent the possible settings of the w Boolean variables on each level of the circuit. By our assumption, we access only a single input variable and thus the new state depends only on that variable and the old state.

The simulation in the other direction is less straightforward. It is easy to simulate a w-BP by a $2w$-circuit, or even a $w+2$-circuit, by storing the branching program state in unary, i.e., in w gates exactly one of which will be on. We can improve matters by storing the state in binary.

Theorem 6: A branching program of width w and length l may be simulated by a Boolean circuit of width $\lceil \log w \rceil + 1$ and length $O(l)$, where the constant depends on w.

Proof: WLOG let $w = 2^m$ be a power of two. To simulate an instruction it suffices to simulate one where either f_i or g_i is the identity, so WLOG we'll assume it's g_i and that the problem is to do f_i if x is on and the identity otherwise.

Note that we need only simulate a set of functions which generates under composition the entire set of functions from $[w]$ to $[w]$ (Here $[w]$ is the set $\{0 \ldots w-1\}$.)

Lemma: The functions from $[w]$ to $[w]$ are generated by: (1)the transpositions f_i, for $0 \leq i < m$, defined by $f(0) = 2^i$, $f(2^i) = 0$, and $f(j) = j$ otherwise; (2) the permutations g_i for $0 \leq i < m$ defined by $g_i(j) = j + 2^i$ for $j < 2^i$, $g_i(j) = j - 2^i$ for $2^i \leq j < 2^{i+1}$, and $g_i(j) = j$ otherwise; and (3) the function h defined by $h(0) = 0, h(1) = 0$, and $h(j) = j$ otherwise.

Proof: We will show that the f_i and g_i generate the permutations of $[w]$, by induction on m. This will suffice, as any function which is not one to one may easily be made up out of permutations and copies of h. The permutations of $[2]$ are clearly generated by f_0. We must show how to generate any permutation of $[w] = [2^m]$, assuming that the f_i and g_i for $i < m-1$ generate all permutations of $[w/2]$. By conjugation with g_{m-1}, we can make all permutations of the elements $\{w/2, \ldots, w-1\}$. Using these permutations as necessary among the high-numbered and low-numbered elements as necessary, we can use f_{m-1} to swap highs for lows as necessary to generate an arbitrary permutation of $[w]$.

Proof of Theorem 6: We will encode the state by m bits $L_0, L_1, \ldots, L_{m-1}$ with the state encoded being $\Sigma_i L_i 2^i$.

We will now view f_i and so forth as the function of x which is the old f_i if x is on and the identity if it is off.

Each f_i or g_i is $L := L \oplus y$ for an appropriate y which is an AND of x and other L's. This is doable using one extra node along with the first m, as follows. First compute $\bar{y}$ using successive ORs, maintaining the L_i's. Then AND $\bar{y}$ with L_i and save the result. Now, using the space for L_i, compute $\bar{L}_i \wedge y$ by a NOT and successive ANDs. As $L_i \oplus y = (L_i \wedge \bar{y}) \vee (\bar{L}_i \wedge y)$, we can now get the new L_i with one OR step. The circuit below illustrates this method, computing the transposition f_2 or (0 4) with $m = 3$ in width 4.

The function h changes L_0 by the assignment $L_0 := L_0 \wedge \bar{y}$, where y is the OR of $\bar{x}$ and all the other L_i's. The other L_i's are not changed. This is easily doable in width $m + 1$, by using one extra column to compute y by successive ORs, complementing it, and then ANDing it in at the end.

Comparing this result with that of [Ho83], we see that our definition of BP width leads to a closer relationship between BP width and circuit width than does the [BDFP83] model. We conclude by summarizing the main consequence of Theorem 6 for bounded-width circuit complexity.

Corollary: The class of languages recognizable by circuits of constant width and polynomial size equals the class of those recognizable with width 4 and polynomial size, as both are NC^1.

9. Non-solvable PBP's and Completeness

The natural question to ask about the proof of Theorem 5 is what properties of the group S_5 were necessary to carry it out. The answer is simply non-solvability, as we will now show. Thus if the Conjecture of Section 6 is true, the languages recognized by poly-size G-PBP's are all of NC^1 if and only if G is not solvable.

Theorem 7: The word problem for any fixed non-solvable group G is complete for NC^1 under AC^0 reductions.

Proof: Without loss of generality, assume that G's commutator subgroup is itself. We show that given a fan-in 2 circuit of depth d and an element a of G not equal to the identity, there is a G-PBP of length at most $(4g)^d$ which yields a if the circuit accepts the input and yields the identity otherwise. Here g is the order of G, a constant. Evaluating a G-PBP is easily seen to be in AC^0, given oracle nodes for the word problem for G. This will suffice to show completeness – the word problem is clearly in NC^1 as we can multiply two permutations in constant size and depth with fan-in two.

The proof, like that of Theorem 5, is by induction on d. The element a must have a representation as a product of at most g commutators. We carry out the proof of Theorem 5, except that we use the inductive hypothesis to produce G-PBP's yielding arbitrary non-identity elements of G instead of five-cycles. This multiplies the length by at most $4g$ instead of 4 at each step. Lemma 1 is unnecessary as for each d, we simultaneously prove the result for all a in G except the identity.

It is interesting to have complete languages for NC^1 which are defined algebraically. The class of complete languages, of course, gives us another new starting point for studying the structure of NC^1 under AC^0 reductions.

10. Uniformity

Before we state and prove a uniform version of Theorem 5, we must review some background. We define an ***alternating Turing machine*** to be a game played by two players on a nondeterministic Turing machine which has two possible state transitions in every position. States are labelled White or Black as to which player has control of the moves from that state. For defining the class $ATIME(\log n)$, we assume that the machine has a random-access input tape of length n (which it can access only once at the end of the computation), a worktape of size $c \log n$ for some constant c, and a clock which restricts it to running for $c \log n$ steps. The players, who are assumed to be omniscient, direct the computation of the machine until the end, when White wins iff he can correctly predict the input bit to be read. The alternating Turing machine is said to accept an input $\mathbf{x}$ iff White has a winning strategy for this game with input $\mathbf{x}$. By standard methods these assumptions may be shown to be perfectly general.

Ruzzo [Ru81] defines NC^1 circuits as those fan-in 2 depth $O(\log n)$ circuits whose extended connection language is in $ATIME(\log n)$. The extended connection language consists of strings of the form $\langle g, h, s\rangle$ where g and h are names of nodes in the circuit, $s \in \{left, right\}^{\leq \log n}$, and h is the node reached by following the path s from g. This has the consequence that $NC^1 = ATIME(\log n)$. We would like to show that the class of languages recognized by $ATIME(\log n)$-uniform polynomial-size bounded-width branching programs is also $ATIME(\log n)$. This will show that $BWBP = NC^1$ in the uniform as well as in the non-uniform setting.

Theorem 8: A language A is in $ATIME(\log n)$ iff it is recognized by a branching program B, of constant width and polynomial size, for which the language:

$$\{\langle k, f, g, i\rangle : \text{the } k\text{'th instruction of } B \text{ yields}$$
$$\text{function } f \text{ if } x_i \text{ is on and } g \text{ if } x_i \text{ is off}\}$$

is in $ATIME(\log n)$.

Proof: First we define a game in which White tries to prove that $B(\mathbf{x}) = f$, for some accepting f, and Black tries to refute him. At each stage of the game the log-time machine will define a range of instructions in B and a function which White claims is yielded by that range. White advances his claim by naming two functions g and h, with $f = gh$, and claiming that the first half of the range yields g and the second h. Black must choose one of these two subclaims to challenge, and this becomes White's new claim for the next stage. After $O(\log n)$ stages White will be making a claim about a single instruction, and this can be verified in $ATIME(\log n)$ by hypothesis. Each stage takes constant time, as we can let Black's sequence of choices be the index of the instruction to be checked – so each bit of this index need only be written down once.

For the converse, given a log-time machine M and game rules to make it an alternating machine, we can get an NC^1 circuit C in a standard way by creating a node for each configuration of M. Let B be the 5-PBP with output (12345), say, created from C by the method of Theorem 5 above, so that B five-cycle recognizes A. We must show that B is $ATIME(\log n)$ uniform. We define a game with input $\langle k, \sigma, \tau, i\rangle$ which White can win iff the input is a correct description of the k'th instruction. Both players, of course, know the actual circuit C and branching program B, as these are uniquely defined from M.

White at each stage will maintain a claim of the following form:

$$\langle s, \mu, k, \sigma, \tau, i \rangle$$

meaning 'The subcircuit C_s of C whose top node is M-configuration s corresponds to a section B_s of B which five-cycle recognizes the language accepted by C_s with output μ. Further, the k'th instruction of B_s yields σ if x_i is on and τ if x_i is off.'

White will begin by claiming $\langle start, (12345), k, \sigma, \tau, i \rangle$ and refine this through $O(\log n)$ moves, each move corresponding to a step of M or to moving down one edge of C. For example, if s is an and-node B_s consists of four sections – White must state in which section the k'th instruction occurs, what its new number is, and which of s's children the section represents. Eventually s will be a final configuration of M and White's claim can be quickly decided. Black's moves during this process are to challenge any White claim which does not follow from his previous claim according to the definition of M and the procedure for creating B. Such a challenge may be decided easily in log-time, ending the game. White's moves are each only a constant number of steps if we choose an appropriate representation for the number k and don't have to rewrite it every time.

It should be clear that this proof will work for other notions of uniformity as well, as we only required that at least the power of the class $ATIME(\log n)$ be available to carry out the simulations in each direction. In particular. log-space or poly-time uniform bounded width branching programs calculate exactly log-space or poly-time uniform NC^1 respectively.

11. Open Problems

We now know that poly-size bounded-width BP's give NC^1 while poly-size general BP's give L. Certainly this suggests a new attack on the problem of whether $NC^1 = L$ as this can now be phrased entirely in terms of branching programs. We also have another new phrasing in terms of bounded width circuits — we would have to show that width $O(\log n)$ is more powerful than width 4, given polynomial size. It would be useful to develop a lower-bound technology for width 5 PBP's or width 4 circuits, if this is possible. Even a superpolynomial lower bound for, say, the clique function would give prove NC^1 different from NP.

The power of general poly-size permutation BP's (no restriction on width) was mentioned as an open problem in [Ba86a]. Cook and McKenzie [CM86] have just shown that the word problem for S_n is complete for log space under NC^1 reductions, even if the inputs and outputs are in pointwise notation (i.e., a permutation σ is given as the list of integers $\sigma(1), \ldots, \sigma(n)$). (In fact, they show that the easier problem of permutation powering with the exponent in unary is complete.) A poly-size PBP can be constructed to solve this problem, given an appropriate definition of recognition of a language by a PBP. As these PBP's can be thought of as *reversible* non-uniform log-space Turing machine computations, this suggests a comparison with work of Bennett [Be73].

The effect of non-determinism on these classes must be examined as well, suggesting possible new attacks on the problem of whether $L = NL$. One must be careful with definitions here, as the wrong sort of non-determinism can turn a very small class into NP. For example, depth-2 poly-size unbounded fan-in Boolean circuits can only recognize Π_2-$TIME(\log n)$. But if we give such a circuit both **x** and **y** inputs and say that it 'accepts' **x** iff there is some **y** such that the circuit accepts $\langle \mathbf{x},\mathbf{y} \rangle$, it can recognize any language in NP.

We know the power of width 3 [Ba85] and width 5 PBP's – what of width 4? As S_4 is solvable, they cannot do all of NC^1 by the method used here for width 5, but we would like to prove they cannot do it at all. The conjecture of Section 6 would settle this, but 4-PBP's are a special case which might be more amenable to analysis.

We know that BP's without the permutation restriction require width 3 to do majority in poly-size [Ya83] and we know that width 5 suffices. Does the extra freedom to use non-permutation instructions help at all?

Circuits of width 2 or 3 are an attractive target for a lower bound proof — it would be nice to prove that width 4 is necessary to do NC^1, if it is.

Can one improve Theorem 6 on simulating BP's by circuits? Of course any bounded width BP can be simulated in width 4 with a polynomial blowup in length using our main result, but can the simulation be improved while keeping linear blowup? Here it might be easier to simulate $2^m + 1$-BP's in width $m + 1$ than to do 2^m-BP's in width m. If the circuit width is less than $\lceil \log w \rceil$, it would seem that there aren't enough states for a direct simulation — can this be proved?

The fine structure of NC^1 is another good subject for further study. We know only that there are at least two classes (from [FSS81] and [Aj83]) but this is more than is known about most degree theories in complexity theory. Fagin et al. [FKPS84], give many AC^0 reducibilities among symmetric functions, but a new proof technique will be needed to settle the conjecture of Section 6 if it is true. AC^0-reducibility should also be compared with the projection reducibility of Valiant [SV81] in this setting. Majority is AC^0-complete for symmetric functions, but no function is projection-complete for them [GS??]. It is also interesting that an algebraically-defined language such as the word problem should be complete.

The unexpected power of NUDFA's suggests some foundational questions. Placing the power to recognize a language in a program to a very simple machine seems very different than placing it in, say, the state table of a Turing machine. How different is it, and how does it relate to other known models of computation?

12. References

[ABHKST86] M. Ajtai, L. Babai, P. Hajnal, J. Komlós, E. Szemerédi, and G. Turán, 'Two lower bounds for branching programs', Proc. 18th ACM STOC, 1986, to appear.

[Aj83] M. Ajtai, 'Σ_1^1 formulae on finite structures', Annals of Pure and Applied Logic 24 (1983), 1-48.

[An85] A. E. Andreev, 'On a method for obtaining lower bounds for the complexity of individual monotone functions', Dokl. Ak. Nauk. SSSR 282 (1985), 1033-1037 (in Russian). English translation in Sov. Math. Dokl. 31 (1985), 530-534.

[Ba85] D. A. Barrington, 'Width-3 permutation branching programs', Technical Memorandum TM-293, M.I.T. Laboratory for Computer Science.

[Ba86] D. A. Barrington, 'Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1', Proc. 18th ACM STOC, 1986, to appear.

[BCH84] P. W. Beame, S. A. Cook, and H. J. Hoover, 'Log-depth circuits for division and related problems', Proc. 25th IEEE FOCS, 1984, 1-6.

[BCP83] A. Borodin, S. A. Cook, and N. Pippenger, 'Parallel computation for well-endowed rings and space-bounded probabilistic machines', Information and Control 58 (Jan. 1983), 113-136.

[BDFP83] A. Borodin, D. Dolev, F. E. Fich, and W. Paul, 'Bounds for width two branching programs', Proc. 15th ACM STOC, 1983, 87-93.

[Be73] C. H. Bennett, 'Logical reversibility of computation', IBM Journal of Research and Development, 17 (1973), 525-532.

[CFL83] A. K. Chandra, M. L. Furst, and R. J. Lipton, 'Multiparty protocols', Proc. 15th ACM STOC, 1983, 94-99.

[CM86] S. A. Cook and P. McKenzie, 'Problems complete for deterministic logarithmic space', Publication 560 (Fév. 1986), Dépt. d'I.R.O., Université de Montréal.

[Co85] S. A. Cook, 'The taxonomy of problems with fast parallel algorithms', Information and Control 64 (Jan. 1985) 2-22.

[FKPS84] R. Fagin, M. M. Klawe, N. J. Pippenger, and L. Stockmeyer, 'Bounded depth, polynomial-size circuits for symmetric functions', IBM Report RJ 4040 (45198) (October 1983), IBM Research Laboratory, San Jose.

[FSS81] M. Furst, J. B. Saxe, and M. Sipser, 'Parity, circuits, and the polynomial time hierarchy', Proc. 22nd IEEE FOCS, 1981, 260-270.

[GS??] M. Geréb-Graus and E. Szemerédi, 'There are no p-complete families of symmetric Boolean functions', preprint.

[Hå86] J. Håstad, 'Improved lower bounds for small depth circuits', Proc. 18th ACM STOC, 1986, to appear.

[Hå86a] J. Håstad, 'Computation limitations for small depth circuits', Ph.D. thesis, Dept. of Mathematics, M.I.T., June 1986.

[Ho83] H. J. Hoover, 'Characterizing bounded width', manuscript, 1983.

[Jo86] D. S. Johnson, 'The NP-completeness column: An ongoing guide', Journal of Algorithms 7:2 (June 1986), to appear.

[LF77] R. E. Ladner and M. J. Fischer, 'Parallel prefix computation', Proc. 1977 Intl. Conf. on Parallel Processing, 218-233.

[Le59] C. Y. Lee, 'Representation of switching functions by binary decision programs', Bell System Technical Journal 38 (1959) 985-999.

[Ma76] W. Masek, 'A fast algorithm for the string editing problem and decision graph complexity', M.Sc. thesis, Dept. of E.E.C.S., M.I.T., May 1976.

[Pu84] P. Púdlak, 'A lower bound on complexity of branching programs', Proc. Conference on the Mathematical Foundations of Computer Science, 1984, 480-489.

[Ra85] A. A. Razborov, 'Lower bounds on the monotone complexity of some Boolean functions', Dokl. Ak. Nauk. SSSR 281 (1985), 798-801 (in Russian). English translation in Sov. Math. Dokl. 31 (1985), 354-357.

[Ra85a] A. A. Razborov, 'A lower bound on the monotone network complexity of the logical permanent', Mat. Zametki 37 (1985) 887-900 (in Russian). English translation in Math. Notes of the Academy of Sciences of the USSR 37 (1985), 485-493.

[Ru81] W. L. Ruzzo, 'On uniform circuit complexity', Journal of Computer and System Sciences 22:3 (June 1981), 365-383.

[Sa72] J. Savage, 'Computation work and time on finite machines', Journal of the ACM 19 (1972), 660-674.

[Sh85] J. B. Shearer, personal communication, 1985.

[Sp71] P. M. Spira, 'On time-hardware complexity tradeoffs for Boolean functions', Proc. 4th Hawaii Symposium on System Sciences (North Hollywood, Calif., Western Periodicals Co., 1971), 525-527.

[SV81] S. Skyum and L. G. Valiant, 'A complexity theory based on Boolean algebra', Proc. 22nd IEEE FOCS, 1981, 244-253.

[Ya83] A. C. Yao, 'Lower bounds by probabilistic arguments', Proc. 24th IEEE FOCS, 1983, 420-428.

[Ya85] A. C. Yao, 'Separating the polynomial-time hierarchy by oracles', Proc. 26th ACM STOC, 1985, 1-10.

[Za58] H. J. Zassenhaus, *The Theory of Groups*, 2nd ed. (New York, Chelsea Publ. Co., 1958).

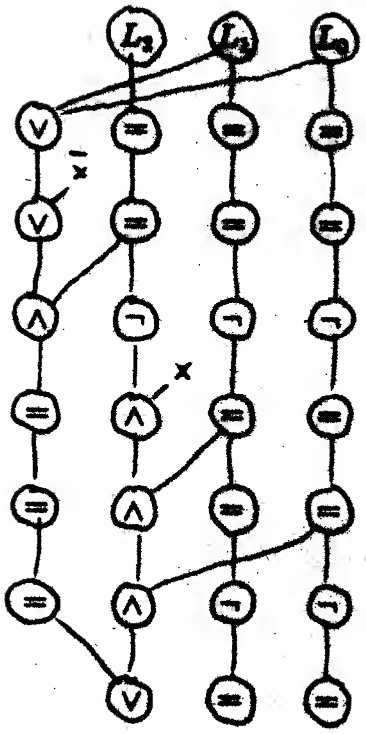

Figure 1: A width 4 circuit calculating f_2.

Bounded Width Branching Programs

David A. Barrington[1]
Department of Mathematics
and Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

16 May 1986

Abstract:

We examine the branching program model of computation and in particular the classes of languages which can be recognized when the width of the programs is bounded by a constant. After slightly revising the framework of definitions to sharpen analogies with other models, we prove that width 5 polynomial size branching programs can recognize exactly the parallel complexity class NC^1, refuting a conjecture of Borodin et al. in [BDFP83]. Other results include an application to Boolean circuits of constant width (here, width 4 and polynomial size circuits can recognize exactly NC^1) and a characterization of a restricted class of width 3 branching programs. This thesis contains the results of [Ba85] and [Ba86], along with some additional material.

Key Words and Phrases:

Branching programs, parallel complexity, circuit complexity.

[1]This work was supported by NSF grant MCS-8304769 and US Air Force grant AFOSR-82-0326, and by an NSF Graduate Fellowship.